

A JavaGeeks.com  
White Paper

Ted Neward  
<http://www.javageeks.com/~tneward>  
tneward@javageeks.com  
22January2002

---

# Self-Modifying Websites

---

Creating Truly Dynamic Webpages

### Abstract

Within the vast realm of Computer Science, one truism has held across all languages and environments: "There is no problem that cannot be solved by adding another layer of indirection." Within the dynamic web-page environment, this can be realized by creating dynamic pages that in turn generate other pages, based on user actions, for the user to then view and use. This paper discusses how it can be done in the Java Servlets/JSP environment, and shows code from a working JSP site.

### Problem discussion

In the beginning, there was static HTML. And it was good. For a fledgling Internet community, the ability to simply display hyperlinked text and graphics and view these "web pages" remotely was enough. Interactivity relied upon navigating through the links, simply absorbing the material presented.

After a while, however, some intrepid engineers decided that it would be useful if the user could somehow interact with the pages in a more active way--by passing in input, gathered perhaps from user-input elements on the web page itself (what we later came to call "form controls") that would be submitted to some programming code on the server. That program would be responsible for generating the resulting response, which frequently was generated in some way based on the user's input. Thus was the Common Gateway Interface born, and the slew of technologies that followed and improved upon it--ISAPI/NSAPI, Active Server Pages, Servlets, Java Server Pages, and most recently, ASP.NET. At first these programs were interpreted (Perl CGI scripts, later ASP), but performance issues led programmers to seek faster solutions, leading to compilation of the programs into native executables (ISAPI, Servlets, ASP.NET).

All of these rely fundamentally upon the same idea--the programmatic constructs within the recipient program (DLL in ISAPI/NSAPI, script code in ASP, Java in Servlets/JSP, or whatever .NET language is used when building the ASP.NET page) examine the input, and decide what to send back based on that data. The programmatic logic decides what response should be generated, and sends that response back to the user.

Problem is, this model assumes that the user is passing in volatile data frequently, usually new data on each request. As web applications have matured, this has become less of a fixed assumption. In fact, it's not uncommon for certain parts of the web app to be "dynamic" in the sense that the page's contents will differ from one user to another, but remain "static" in that that particular user's page-settings remain constant for a period of time stretching over days, weeks, months or even years.

Consider a simple example: A simple information-portal application. A user signs up with the portal service, and is immediately asked to select what kinds of information he wishes to see on his home page at that portal. In the typical scenario, the user's preferences are stored off into a database. The next time the user enters the portal and moves to his home page, the generic home page script must retrieve the user's individual settings to determine what information the user wants on the page, then retrieve that information and render it accordingly.

Take note of something very important here--the home page for the user is the same generic home page for all users, with appropriate code to dynamically generate output based on the user's chosen settings or data. This is important because the user's chosen settings will change rarely, if ever--few users of a portal change the kinds of information they're interested in, and even among those who do, it doesn't occur more than infrequently.

Despite the fact that the criteria by which the decisions are made remains relatively static, the pages are still entirely dynamic--each time a given user retrieves his home page, the same result must be recalculated again and again and again. This is not to imply that all of the data is static--the user's preferred data items may be changing on a daily or even hourly basis--but that certain decision points will always flow the same way on the page, such as the user's choices of categories.

At this point in the discussion, it's fair to ask precisely why this happens--if this is such an obvious and undesirable hit, then why not create a page for that user and be done with it? The answer is equally obvious--developers have neither the time nor the inclination to spend all of their days developing uniquely personalized portal home pages on behalf of individual users. Worse, the idea that programmers could do so simply won't scale as the portal gains in popularity. And recall that the portal idea is simply one of several possible places where a given set of criteria do not change frequently. Another is the act of generating reports against a database. Yet another is the idea of allowing users of a web system to influence the web system's content in some way, quite similarly to a WikiWikiWeb. In each of these cases, asking a developer to make necessary changes on the page is unrealistic.

In many respects, this is beginning to approach a new dimension on web page design; at first, we had static web pages. Then pages became dynamic. Now, we wish pages to exude both tendencies--dynamism in order to execute the (statically-known) same set of logic, yet dynamically able to change that logic without developer intervention. In short, we want the ability, for certain applications, to modify the "source code" of the page without developer intervention.

### Solution discussion

**Self-modifying executable pages.** The ability to modify "source code" on the fly is definitely something that bears some technical challenges. Consider the traditional programmatic solution for building a "dynamic" web page: the programmer sits down at her desk, writes some code, runs it through a compiler (fixing typographic or semantic errors as necessary) to produce the executable bits, then installs the executable into the hosting environment for testing/execution.

Traditionally, the only "computer-generated" code that ever fit into this equation was that of language "wizards" which would generate source code once, based on developer inputs, and developers would modify the generated source as necessary. Once generated, then, the code was never computer-generated again, and certainly not done during live production runs of the system<sup>1</sup>.

While it's certainly feasible to do, the idea of generating source, executing the compiler, loading and executing the results, and maintain the ability to do this multiple times within a single process space is usually more work than most developers are willing to take on, regardless of the potential benefits. Fortunately, such scaffolding already exists, in the form of the source-based "translation engines" currently forming the backbone of dynamic web page development: ASP, ASP.NET, and JSP. For this paper, we will be examining the abilities of Servlets and JSP <sup>2</sup> to create self-modifying websites.

For starters, let's consider what has to happen in order to build a self-modifying website. Every time the code for the page is modified, the page must be recompiled and redeployed into the hosting environment. While it would be possible to regenerate servlet code and invoke "javac", compiling directly somewhere into the servlet container's Java classpath, a much better mechanism exists, that of modifying the JSP page directly. Consider the following servlet:

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class GenerateJSP
{
    public void doGet(HttpServletRequest request, HttpServletResponse response)
    {
        FileWriter fw = new FileWriter(getServletContext().getRealPath("/generated.jsp"));
        PrintWriter pw = new PrintWriter(fw);
        pw.println("<html>");
        pw.println("<head><title>Generated JSP page</title></head>");
        pw.println("<body>");
        pw.println("This JSP page was generated on " + new java.util.Date());
        pw.println("Click <a href=\"\" + request.getContextPath() + \"/servlet/GenerateJSP\">here</a>
to regenerate me.");
    }
}
```

```
pw.println("</body>");
pw.println("</html>");
pw.close();
fw.close();
pw = new PrintWriter(response.getWriter());
pw.println("<html>");
pw.println("<head><title>Welcome</title></head>");
pw.println("<body>");
pw.println("<a href=\" + request.getContextPath() + \"/generated.jsp\">generated.jsp</a>");
pw.println("</body>");
pw.println("</html>");
pw.flush();
}
}
```

### Modifying a JSP page

Each time the HTML browser points to <http://localhost/selfmodifying/servlet/GenerateJSP>, a JSP file called "generated.jsp" will be created and populated with a simple HTML page displaying the date and time the JSP page was created. When the hyperlink to "generated.jsp" is clicked, the normal recompile-on-demand semantics of JSP kick in, the generated.jsp file is passed through the JSP engine to be translated to Java source, compiled, and finally executed. The net result is the static output "This JSP page was generated on Sun Feb 03 01:33:48 PST 2002<sup>3</sup>. Click here to regenerate me.", with "here" hyperlinked back to the GenerateJSP servlet. Note that unlike a standard JSP page with a `java.util.Date` object embedded in it, the date displayed on the JSP page is fixed; refreshing the "generated.jsp" page will yield the same date every time. Only by re-executing will the date change, because the servlet will then regenerate the JSP page.

As another example, consider a servlet (not displayed here) that presents a simple HTML form containing two form fields: a textbox and a file-upload field. The textbox contains the name of a .jsp file, and the file-upload field will be the file to be copied and deployed to the servlet container. The servlet, when executed, pulls in the submitted file content, writes it to the .jsp file specified by the form field textbox, and we have in essence created a simple distributed authoring system for developers without FTP access to post new pages to the server without having to go through filesystem access. (This could also be used to submit new binary-compiled servlets, assuming the servlet container supports reload-on-demand for servlets, as well.) While not a complete replacement for a WebDAV-based system, it serves as a cheap and simple deployment system for developers behind firewalls using an ISP to host their JSP pages.

The above two examples are fairly convoluted examples, however, and a fair question to ask at this point is whether this approach has any "real" application or use. One such use is that of a "Wiki Clone", a user-modifiable collection of web pages used as a discussion forum. (For more information on the original WikiWikiWeb, see <http://c2.com>.)

**Example: A WikiWikiWeb.** A WikiWikiWeb is a set of webpages, modifiable by users of the Web, allowing for readers of the pages to in turn make comments and influence the discussion, in a more free-form manner than traditional message boards like Slashdot. Pages within the Wiki are referenced using simple formatting; in the original Wiki, simply `RunningCapitalWordsTogether` in the text was enough to generate a hyperlink in the HTML to another page. If that page didn't exist, the hyperlink would be a question mark right after the topic name--so if `RunningCapitalWordsTogether` doesn't exist as a page, then it would appear as "RunningCapitalWordsTogether?", with the question mark itself hyperlinked, rather than the entire word. This allows for easy creation of new links. Each page usually contains a hyperlink or button to take the user to a "Edit this Page" form, where the original text of the page can be edited, deleted and/or appended. New text is then saved to the topic page when the Edit form is submitted. Text formatting is done using simple non-HTML sequences; for example, in the original Wiki, emphasis (bold) is indicated by wrapping the desired text in two single-quote characters on either side: "This would be bold". More information on Wikis can be found in [WikiWay] .

Most Wikis are implemented using a simple separation between the storage of the user-entered text and

the rendering logic to transform the text into the appropriate HTML. Frequently, the user text is stored in a relational database of some form, retrieved based on the topic being browsed, and rendered on-demand. Given that perhaps only one in twenty people browsing a given Wiki page will actually modify the page, the continuous re-fetch of the text from the database is a performance and scalability waste--if the data hasn't changed, why should we have to retrieve it over and over again?

In this WikiClone<sup>4</sup>, which I call JSPWiki, instead of storing the user text to a central database, I instead store the user text directly within JSP pages, one per topic. The name of the topic corresponds directly to the .jsp page containing that text, and user modifications to the user text in turn will regenerate the JSP page for that topic.

The heart of the system centers around two servlets<sup>5</sup>, Edit and Save, which provide the central page-modification logic. The Edit servlet, when fired, must load the existing JSP user text and present that in a form for user modification. This is not necessarily an easy task, since there will be "decorations" stored in each JSP page that aren't user text--last modified date/time, for example, as well as any header and/or footer text above and below the user text. The Save servlet, then, is invoked from the Edit servlet, and is responsible for taking the user text, saving it to the JSP page, and forwarding to that JSP page.

In the interests of clean separation of concerns, no text-rendering logic is performed in the JSP page; instead, that is deferred to a JSP custom tag-handler whose implementation will not be shown here. Its operation is not important (except as a discussion of Wiki formatting choices and rules) to the topic at hand.

The Save servlet is perhaps the easier of the two to understand, so I begin by showing that code first:

```
import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;
/**
 * Servlet to save off the edited wiki page, by modifying the JSP file.
 */
public class Save extends HttpServlet
{
    public void doPost(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException
    {
        ServletContext application = getServletContext();
        FileWriter fw = new FileWriter(application.getRealPath(request.getParameter("pagename")));
        PrintWriter pw = new PrintWriter(fw);
        String text = request.getParameter("text");
        Date now = new Date();
        String pagefile = request.getParameter("pagename");
        String pagetitle = getTitle(pagefile);
        //
        // Header
        //
        pw.println("<%@ taglib uri=\"http://www.javageeks.com/taglibs/jspwiki-1.0\" prefix=\"wiki\"
%>");
        pw.println("<html>");
        pw.println("<head><title>" + pagetitle + "</title></head>");
        //
        // Style
        //
        pw.println("<%@ include file=\"WEB-INF/style.css\" %>");
        //
        // Text
        //
    }
}
```

```
pw.println("<body>");
pw.println("<h1>" + pagetitle + "</h1>");
pw.println("<table width=\"100%\" cellpadding=\"8\" cellspacing=\"0\">");
pw.println("<tr style=\"background-color: white\"><td>");
pw.println("<wiki:formatter>");
pw.println(text.trim());
pw.println("</wiki:formatter>");
pw.println("</td></tr>");
//
// Footer
//
pw.println("<tr><td>");
pw.println("<a href=\"<%= request.getContextPath() %>/servlet/Edit?<%=
request.getServletPath() %>\>Edit this page</a> | ");
pw.println("<a href=\"<%= request.getContextPath() %>\>Home page</a> | ");
pw.println("<a href=\"<%= request.getContextPath() %>/servlet/Modified\">Modified pages</a>
| ");
pw.println("<a href=\"<%= request.getContextPath()
%>/servlet/Search?pattern=&find=find+in+name\">List all pages</a>");
pw.println("<br>");
pw.println("<form method=\"get\" action=\"<%= request.getContextPath()
%>/servlet/Search\">");
pw.println("<input type=\"text\" name=\"pattern\">");
pw.println("<input type=\"submit\" name=\"find\" value=\"find in name\">");
pw.println("<input type=\"submit\" name=\"find\" value=\"find in content\">");
pw.println("</form>");
pw.println("<br>");
pw.println("<i>Last modified on " + now + "</i>");
pw.println("</td></tr></table>");
pw.println("</body>");
pw.println("</html>");
pw.flush();
fw.close();
//
// Make a note in the _modified.jsp pages
//
String[] list = new String[20];
try
{
    FileReader fr = new FileReader(application.getRealPath("WEB-INF/_modifiedfiles.jsp"));
    BufferedReader br = new BufferedReader(fr);
    for (int i=1; i<list.length; i++)
    {
        String line = br.readLine();
        if (line == null)
            break;
        else
            list[i] = line;
    }
    fr.close();
}
catch (IOException ioEx)
{
    // Eat it--the file didn't exist, so we'll just create a new one now
}
list[0] = "<a href=\"\" + request.getContextPath() + pagefile + \"\">" + pagetitle + "</a> by
" +
    request.getRemoteHost() + " on " + now + "<br>";
FileWriter modfw = new FileWriter(application.getRealPath("WEB-INF/_modifiedfiles.jsp"));
```

```
PrintWriter modpw = new PrintWriter(modfw);
for (int i=0; i<list.length; i++)
    if (list[i] != null)
        modpw.println(list[i]);
modpw.flush();
modfw.close();
//
// Forward back to the page in question
//
application.getRequestDispatcher(pagefile).forward(request, response);
}
/**
 * Utility method to get the topic out of the JSP filename.
 */
private String getTitle(String pagename)
{
    // Strip out the first / and the last .jsp
    //
    return pagename.substring(1, pagename.length()-4);
}
}
```

### Save

The servlet itself can be broken into three key parts: generating the .jsp file, making a note in a "most-recently-modified" .jsp page (for doing "What's New"-style lists), and forwarding back to the recently-modified topic page. As can be seen, none of the code is particularly complex; perhaps the most difficult part of writing this kind of code is ensuring that all quoted strings are appropriately quote-escaped in the servlet. Note that the user text itself is coming in a form field called "text", and is written out in a single `println` call; the rest of the text is all header and footer JSP code.

This header and footer JSP code is what's going to make the Edit servlet interesting--it needs, somehow, to extract only the user text from the JSP page, and ignore the extraneous header and footer code. Fortunately, the custom tag ("`<wiki:formatter>`") begin and end tags serve as useful markers indicating the beginning and ending of the user text, so the Edit servlet uses that to indicate where to start reading and when to stop reading the text from the JSP file. (In addition, the Edit servlet will have to determine if this JSP file even exists, since it could be invoked in order to create the page.) The Edit servlet itself pulls the text in, puts each text line into an `ArrayList`, then puts the `ArrayList` itself into the `request` object and forwards to a JSP page for displaying the form the user uses to edit the text:

```
import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;
/**
 * Servlet to edit (or create?) a wiki topic JSP page.
 */
public class Edit extends HttpServlet
{
    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException
    {
        //
        // Get the incoming .JSP file, if it exists
        //
        InputStream in = getServletContext().getResourceAsStream(request.getQueryString());
        if (in != null)
        {
            ArrayList text = new ArrayList();
```

```
BufferedReader br = new BufferedReader(new InputStreamReader(in));
String line;
while (!(br.readLine().equals("<wiki:formatter>")))
{
    ;
}
while (!((line = br.readLine()).equals("</wiki:formatter>")))
{
    text.add(line.trim() + "\n");
}
request.setAttribute("text", text);
}
else
{
    ArrayList text = new ArrayList();
    request.setAttribute("text", text);
}
//
// Forward to a JSP page to do the HTML out
//
getServletContext().getRequestDispatcher("/WEB-INF/edit.jsp").forward(request, response);
}
}
```

### Edit

The code here is fairly straightforward--try to find the file requested, using the `getResourceAsStream` call, and if it exists, pull in all the text in between the formatter tags. The text lines themselves will go into an `ArrayList`, for display in a `TEXTAREA` HTML control on the subsequent JSP page (`edit.jsp`). If the file doesn't exist, there's no text to pull, so just create an empty `ArrayList`; this way the `edit.jsp` can always assume that a valid `ArrayList` is present (under the name "text") in the `request` object. Notice also that the `edit.jsp` file lives in the `WEB-INF` directory; this prevents the user from accidentally browsing to that `.jsp` page outside of this normal flow of invocation. (Remember, the Servlet specification forbids the User-Agent from being able to see the contents of the `WEB-INF` directory, but servlets and JSPs are perfectly capable of using files there.)

The `edit.jsp` takes the `ArrayList` in `request`, iterates through it, and puts each `String` into a `TEXTAREA` on an HTML form, giving the user the ability to edit the existing text or add further text:

```
<%@page import="java.io.*" %>
<%@page import="java.util.*" %>
<html>
<head><title>Edit <%= request.getQueryString() %></title></head>
<%@include file="style.css" %>
<body bgcolor="#CCCCCC">
<form action="<%= request.getContextPath() %>/servlet/Save" method="post">
<textarea name="text" rows="20" cols="80"><%
for (Iterator i = ((ArrayList)request.getAttribute("text")).iterator(); i.hasNext(); )
{
    out.println(((String)i.next()).trim());
}
%>
</textarea><br>
<input type="hidden" name="pagename" value="<%=request.getQueryString()%>">
<input type="submit" value="Save">
</form>
</body>
</html>
```

### edit.jsp

The HTML form itself is told to POST to the Save servlet, which we saw above.

And, in short, we're done. We have everything in place we need to make a functional Wiki. Granted, there are a few more amenities to add, such as topic search or content search, but these aren't difficult to see how they'd be done: a topic search would simply scan through the directory of .jsp files, looking for filenames that match the search criteria, while a content search would look through the body of the .jsp files themselves <sup>6</sup>.

**Other technical approaches.** It should be fairly obvious to those familiar with any of the other interpretive environments (ASP, ASP.NET, even CGI scripts in Perl) that this technique could be easily adapted to alternative technologies--certainly Java and JSP has no lock on this idea. Remember, however, the one of the key advantages that JSP, and ASP.NET for that matter, have over pure interpretive engines like ASP or Perl is that much is lost doing interpretation of the page over and over again--by compiling the page, we don't have to pay the cost of running the interpreter over the page over and over again.

For that matter, if all the inputs are known at page-generation time, then it's entirely possible to generate the code straight to HTML, rather than to the compiled executable format of Servlets; this would reduce the CPU load on the server significantly, and by extension increase the scalability and performance of the server. Since HTML cannot do any sort of server-side programmatic action, however (such as pull data from a database), generating straight to HTML would only be possible for those pages that are entirely read-only.

Lastly, this sort of same effect can be achieved in an entirely different approach, using XSLT over dynamically-generated (or modified, in the case of a Wiki) XML files. So, for example, when users set up their preferences for their information portal page, those preferences are captured in an XML file, then an XSLT engine is run over the XML file with an XSL file to transform it into the necessary HTML, JSP, or ASP.NET page.

## Consequences

Using this technique carries its share of pros and cons.

For starters, this technique is really only useful for "read-mostly" data, where the page viewed is unchanging except infrequently--the Wiki page satisfies this requirement, as does the information portal page, as does the report page. (In the case of the report, the data being reported may be constantly changing, but which data to retrieve and report isn't.) If the page is entirely read-only, with no user interaction at all possible--a home page, for example, or a menu--then there's no reason to ever regenerate the page programmatically, so this doesn't even become an issue. If the page is entirely mutable, however, such as the report whose columns are generated based on a previous page's settings, then it doesn't make sense to use this, either; once the page had been generated, it will be unlikely that anyone would ever use it again, and so the time taken to generate the page would be a waste.

One advantage of this approach is the fact that it is extremely lightweight--not only is a network round-trip to a database avoided, but the database itself may be entirely done away with in some cases (such as the case of the JSPWiki above). This improves both performance and scalability of the site as a whole.

On top of this, because all the "data" is captured as part of the JSP files on the file system, backup becomes simpler. If up-to-the-second backups are desired, then the Save servlet could store the files into a CVS repository (ASP.NET could use a Visual SourceSafe database, thanks to VSS's COM-based interface model). Alternatively, simply run an "xcopy" to an alternative filesystem on a different machine once a night from a scheduled script or batch file.

There are a couple of points that will need to be considered in using this approach, however. A big one

that's not been addressed anywhere above is that of concurrency--what happens if two people's actions trigger a regeneration of the same JSP file with different settings? Or, perhaps even more importantly, what happens if that file is in use when a regen is fired? Different operating systems will have different rules regarding reader and writer usage of a file, and require different synchronization primitives to control concurrent usage/modification of a file. Unfortunately, none of those primitives or APIs will be available to us in Java without dropping to JNI. For the case of the Wiki, above, it's probably sufficient to assume that "no two Wiki readers will ever modify the same page at the same time", and that if it does happen, "last one in wins" is an acceptable state of affairs. For the case of information portal, again, these rules hold, since we're generating pages on a per-user basis--most users won't have multiple browser windows open, using all of them simultaneously to modify their portal settings.

Additionally, in the case of JSP and ASP.NET, there's the one-time hit of recompilation that will occur on each page modification that has to be factored into the response time. Once again, this just reinforces the idea that this technique really is best suited for "read-mostly" pages, thereby amortizing that one hit across more page accesses. (One slow page load will generally be brushed off by users as "just another slow Internet day".)

### Summary

The compile-on-demand ability of the Java Server Pages technology (and others) allows for some flexibility in architecture design that would be impossible without it. In particular, the ability to generate JSP pages for immediate compilation-and-use in response to user input and action allows for a more flexible, yet still responsive system. Judicious use of this technique can create pages--or sites--that don't require as many trips to the database for content, or even a database at all.

### Notes

- [1] Chris Sells, of DevelopMentor, built a software product called Gen<X> that would "rewizard" generated code into another wizard-based format, but again this was done under developer control, not during production execution.
- [2] Owing simply to the fact that the Java environment is where I first developed the prototype, not out of any inherent advantage over ASP or ASP.NET.
- [3] With the date appropriate to the time the servlet was executed, of course.
- [4] "WikiClone" is the term given to any software system which provides WikiWikiWeb-like behavior.
- [5] They could, and in fact did start out as, be JSP pages in of themselves; remember that JSP is simply another way to write a servlet.
- [6] This is one area where the file-based nature of this Wiki could run into problems--doing this kind of full-scan across every file in the directory could take a while.

### Bibliography

- **[WikiWay]** *The Wiki Way: Quick Collaboration on the Web*, by Bo Leuf, Ward Cunningham. Addison-Wesley (ISBN: 0-201-71499-X)

### Copyright

This paper, and accompanying source code, is copyright © 2002 by Ted Neward. All rights reserved. Usage for any other purpose than personal or non-commercial education is expressly prohibited without written consent. Code is copywrit under the Lesser GNU Public License (LGPL). For questions or concerns, contact author.

### Colophon

This paper was written using a standard text editor, captured in a custom XML format, and rendered to PDF using the Apache Xalan XSLT engine and the Apache FOP-0.17 XSL:FO engine. For information on the process, contact the author at [tneward@javageeks.com](mailto:tneward@javageeks.com). Revision \$Revision: 1.1 \$, using whitePaper.xml revision \$Revision: 1.1 \$.